

Advance Caching

Today

- CAPES!
- quiz 6 + 7 recap
- advanced caching
- Maybe virtual memory
- Hand a bunch of stuff back.

Key Points

- What are the three C's?
- What are 1 or 2 techniques for addressing each of them?
- Be able to simulate a cache by hand.

Speeding up Memory

- $ET = IC * CPI * CT$
- $CPI = noMemCPI * noMem\% + memCPI * mem\%$
- $memCPI = hit\% * hitTime + miss\% * missTime$

- Miss times:
 - L1 -- 20-100s of cycles
 - L2 -- 100s of cycles

- How do we lower the miss rate?

Know Thy Enemy

- Misses happen for different reasons
- The three C's (types of cache misses)
 - Compulsory: The program has never requested this data before. A miss is mostly unavoidable.
 - Conflict: The program has seen this data, but it was evicted by another piece of data that mapped to the same "set" (or cache line in a direct mapped cache)
 - Capacity: The program is actively using more data than the cache can hold.
- Different techniques target different C's

Reducing Compulsory Misses (I)

- Increase cache line size so the processor requests bigger chunks of memory.
- For a constant cache capacity, this reduces the number of lines.
- This only works if there is good spatial locality, otherwise you are bringing in data you don't need.
- If you are asking small bits of data all over the place (i.e., no spatial locality) this will hurt performance
- But it will help in cases like this

```
for (i = 0; i < 1000000; i++) {  
    sum += data[i];  
}
```

Reducing Compulsory Misses (I)

- Increase cache line size so the processor requests bigger chunks of memory.
- For a constant cache capacity, this reduces the number of lines.
- This only works if there is good spatial locality, otherwise you are bringing in data you don't need.
- If you are asking small bits of data all over the place (i.e., no spatial locality) this will hurt performance
- But it will help in cases like this

```
for (i = 0; i < 1000000; i++) {  
    sum += data[i];  
}
```

One miss per cache line worth of data

Reducing Compulsory Misses (2)

- Caches can guess what data the processor will need.
- One technique: Prefetching (240A to learn more)

```
for (i = 0; i < 100; i++) {  
    sum += data[i];  
}
```

- In this case, the processor could identify the pattern and proactively “prefetch” data program will ask for.
- Current machines do this alot...
- **Keep track of** `stride = thisAddress - lastAddress`, **it's consistent**, **start fetching** `thisAddress + stride`.
- The simplest version is a “next line prefetcher”

Reducing Compulsory Misses (2)

- Caches can guess what data the processor will need.
- One technique: Prefetching (240A to learn more)

```
for (i = 0; i < 100; i++) {  
    sum += data[i];  
}
```

- In this case, the processor could identify the pattern and proactively “prefetch” data program will ask for.
- Current machines do this alot...
- **Keep track of `stride = thisAddress - lastAddress`, it's consistent, start fetching `thisAddress + stride`.**
- The simplest version is a “next line prefetcher”

Potentially, one miss per sequence of accesses.

Reducing Compulsory Misses

- Software prefetching
- Use register \$zero!

```
for (i = 0; i < 1000000; i++) {  
    sum += data[i];  
    "load data[i+16] into $zero"  
}
```

Reducing Compulsory Misses

- Software prefetching
- Use register \$zero!

```
for (i = 0; i < 1000000; i++) {  
    sum += data[i];  
    "load data[i+16] into $zero"  
}
```

For exactly this reason, loads to \$zero never fail (i.e., you can load from any address into \$zero without fear)

Conflict Misses

- Conflict misses occur when the data we need was in the cache previously but got evicted.
- Evictions occur because:
 - Direct mapped: Another request mapped to the same cache line
 - Associative: Too many other requests mapped to the same cache line ($N + 1$ if N is the associativity)

```
while(1) {  
    for(i = 0; i < 2000000000; i  
        +=4096) {  
        sum += data[i];  
    } // Assume a 4 KB Cache  
}
```

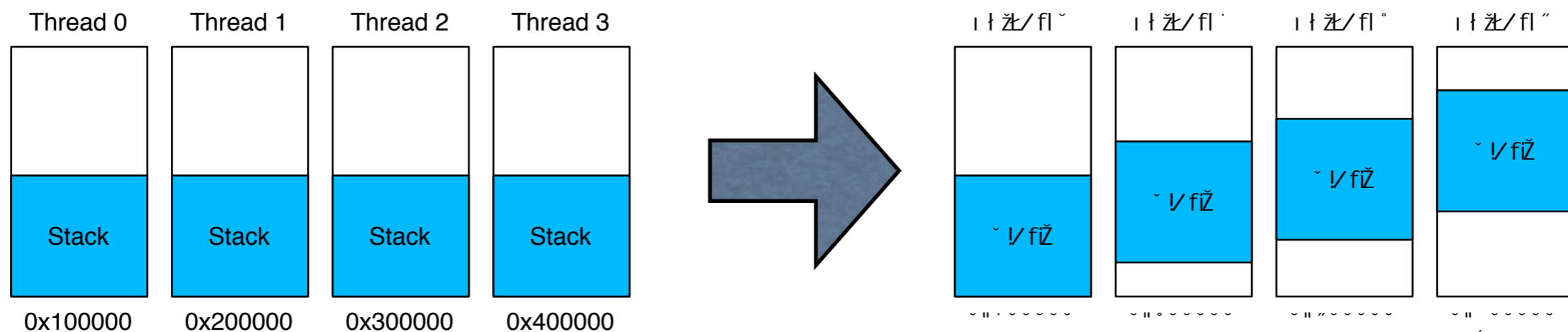
- Every accessed address will have `index == 0`

Reducing Conflict Misses

- Conflict misses occur because too much data maps to the same “set”
 - Increase the number of sets (i.e., cache capacity)
 - Increase the size of the sets (i.e., the associativity)
- The compiler and OS can help here too

Colliding Threads and Data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128MB).
 - Threads often run the same code in the same way
 - This means that thread stacks will end up occupying the same parts of the cache.
 - Randomize the base of each threads stack.
- Large data structures (e.g., arrays) are also often aligned. Randomizing malloc() can help here.



Capacity Misses

- Capacity misses occur because the processor is trying to access too much data.
 - Working set: The data that is currently important to the program.
 - If the working set is bigger than the cache, you are going to miss frequently.
- Capacity misses are bit hard to measure
 - Easiest definition: non-compulsory miss rate in an equivalently-sized fully-associative cache.
 - Intuition: Take away the compulsory misses and the conflict misses, and what you have left are the capacity misses.

Reducing Capacity Misses

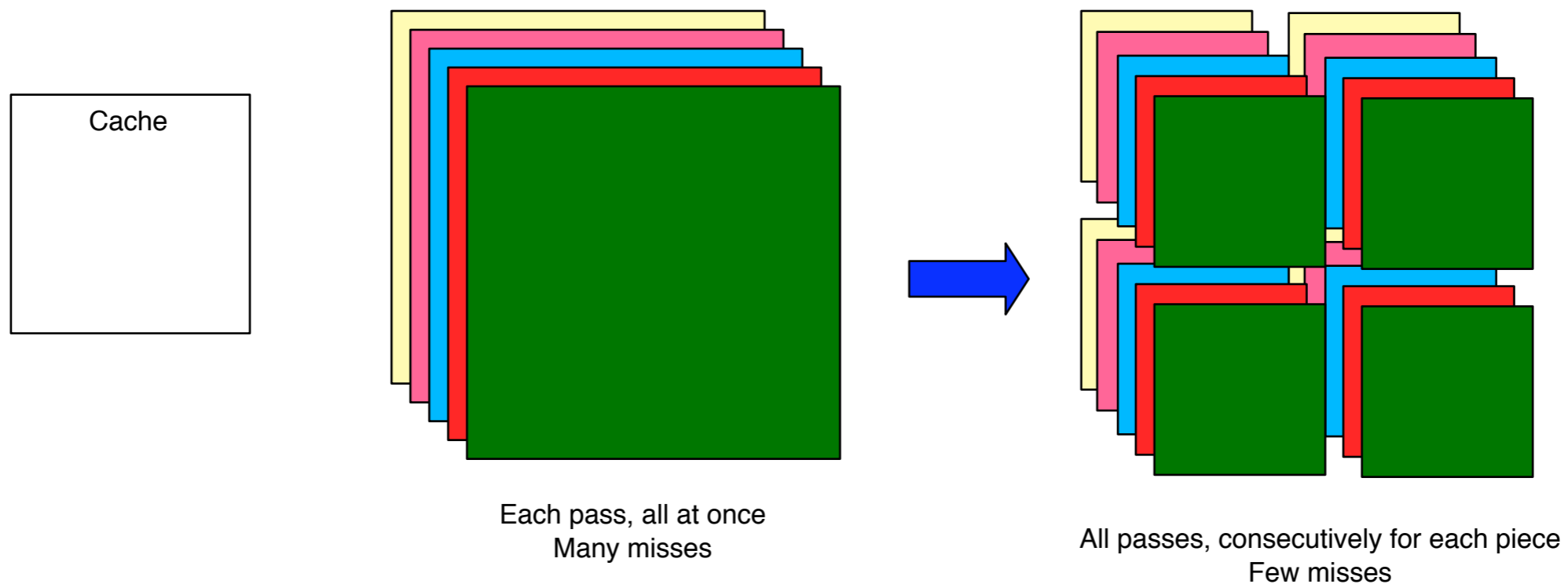
- Increase capacity!
- More associativity or more associative “sets”
- Costs area and makes the cache slower.
- Cache hierarchies do this implicitly already:
 - if the working set “falls out” of the L1, you start using the L2.
 - Poof! you have a bigger, slower cache.
- In practice, you make the L1 as big as you can within your cycle time and the L2 and/or L3 as big as you can while keeping it on chip.

Reducing Capacity Misses: The compiler

- The key to capacity misses is the working set
- How a program performs operations has a large impact on its working set.

Reducing Capacity Misses: The compiler

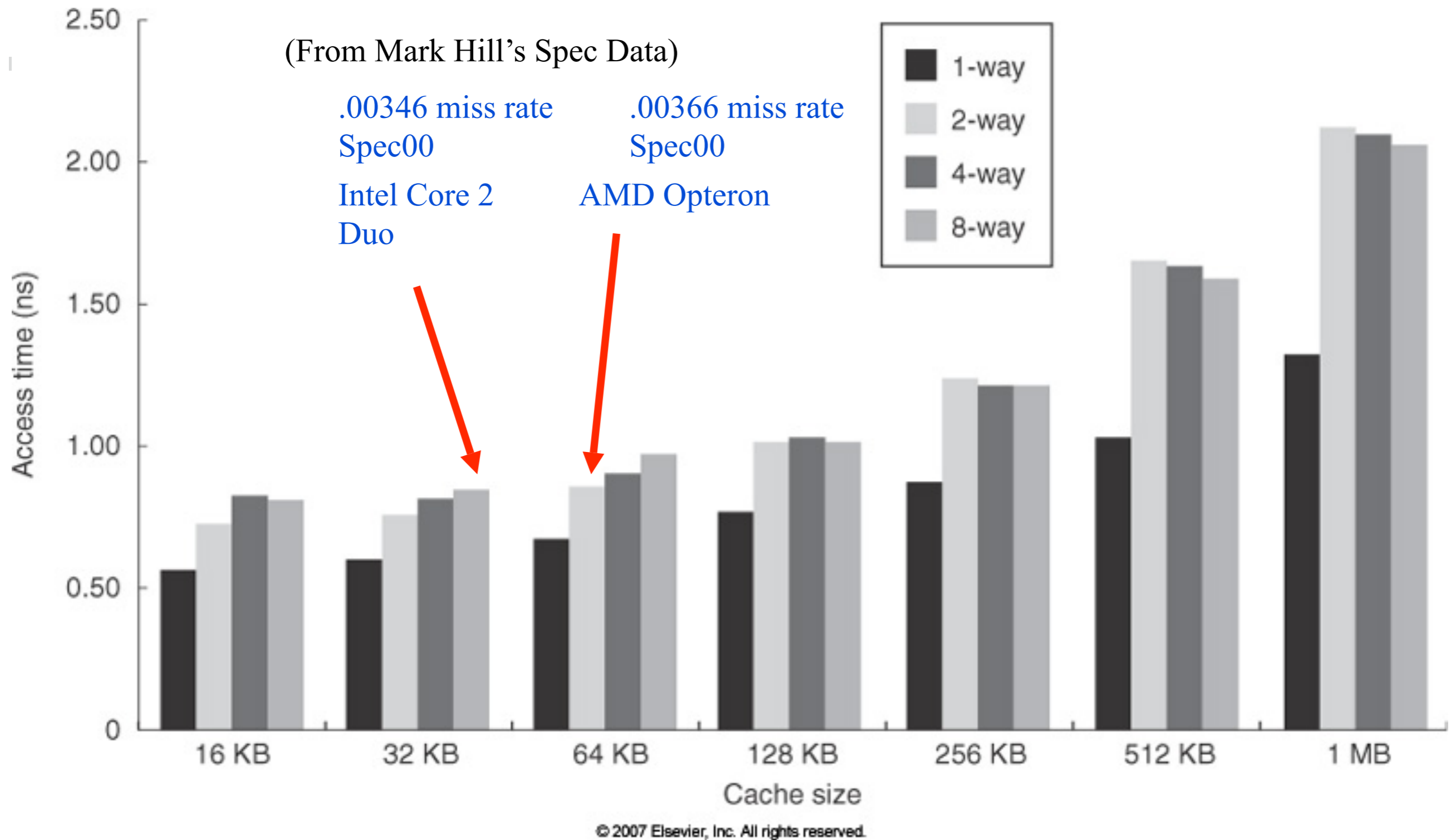
- Tiling
 - We need to make several passes over a large array
 - Doing each pass in turn will “blow out” our cache
 - “blocking” or “tiling” the loops will prevent the blow out
 - Whether this is possible depends on the structure of the loop
- You can tile hierarchically, to fit into each level of the memory hierarchy.



Real Cache Behavior

- Live Demo...The return! Part Deux!
- 32KB, 8-way set associative L1; 32 byte lines
- 4MB, 16-way set associative L2; ?? byte lines

Cache optimization in the real world: Core 2 duo vs AMD Opteron (via simulation)



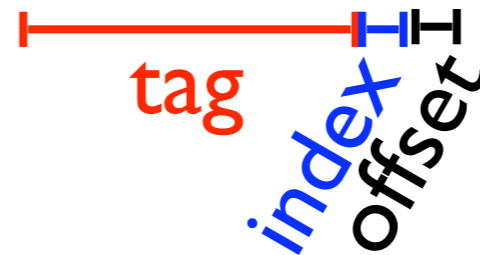
Intel gets the same performance for less capacity because they have better SRAM Technology: they can build an 8-way associative L1. AMD seems not to be able to.

A Simple Example

- Consider a direct mapped cache with 16 blocks, a block size of 16 bytes, and the application repeat the following memory access sequence:
 - 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010

A Simple Example

- a direct mapped cache with 16 blocks, a block size of 16 bytes
- $16 = 2^4$: 4 bits are used for the index
- $16 = 2^4$: 4 bits are used for the byte offset
- The tag is $32 - (4 + 4) = 24$ bits
- For example: 0x80000010



A Simple Example

	valid	tag	data
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	800000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	800000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	800000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	300000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	300000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	300000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	300000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	800000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	800000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010 miss: conflict

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1	1	800000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!


0x80000010 miss: conflict

0x80000018 hit!

A Simple Example: Increased Cache line Size

- Consider a direct mapped cache with 8 blocks, a block size of 32 bytes, and the application repeat the following memory access sequence:
 - 0x80000000, 0x80000008, 0x80000010,
0x80000018, 0x30000010

A Simple Example

- a direct mapped cache with 8 blocks, a block size of 32 bytes
 - $8 = 2^3$: 3 bits are used for the index
 - $32 = 2^5$: 5 bits are used for the byte offset
 - The tag is $32 - (3 + 5) = 24$ bits
 - For example: $0x80000010 =$
 - 
 $01110000000000000000000000010000$
tag index offset

A Simple Example

	valid	tag	data
0			
1			
2			
3			
4			
5			
6			
7			

0x80000000

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	300000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	300000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 miss: conflict

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 miss: conflict

0x80000008 hit!

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 miss: conflict

0x80000008 hit!

0x80000010 hit!

0x80000018

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 miss: conflict

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

A Simple Example: Increased Associativity

- Consider a 2-way set associative cache with 8 blocks, a block size of 32 bytes, and the application repeat the following memory access sequence:
 - 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010

A Simple Example

- a 2-way set-associative cache with 8 blocks, a block size of 32 bytes
- The cache has $8/2 = 4$ sets: 2 bits are used for the index
- $32 = 2^5$: 5 bits are used for the byte offset
- The tag is $32 - (2 + 5) = 25$ bits
- For example: $0x80000010 =$



A Simple Example

	valid	tag	data
0			
1			
2			
3			

0x80000000

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
1			
2			
3			

0x80000000

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
1			
2			
3			

0x80000000 **miss: compulsory**

0x80000008

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
	1	600000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
	1	600000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
	1	600000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
	1	600000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
	1	600000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010 hit!

0x80000018

A Simple Example

	valid	tag	data
0	1	1000000	
	1	600000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

Write Allocate v.s. No - Write Allocate

- Consider the following code

```
int A[1024]; /* A = 0x1000 */
int B[1024]; /* B = 0x2000 */

int i;
for(i = 0 ; i < 1024; i++)
    A[i] = B[i]; /* load B[i], store A[i] */
```

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
-----------	---------	-----	-------	-------------	----------

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	hit	hit

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	hit	hit
store A[1]	0x1004	0x1	0x00	hit	miss

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	hit	hit
store A[1]	0x1004	0x1	0x00	hit	miss
load B[2]	0x2008	0x2	0x00	hit	hit

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	hit	hit
store A[1]	0x1004	0x1	0x00	hit	miss
load B[2]	0x2008	0x2	0x00	hit	hit
store A[2]	0x1008	0x1	0x00	hit	miss

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	hit	hit
store A[1]	0x1004	0x1	0x00	hit	miss
load B[2]	0x2008	0x2	0x00	hit	hit
store A[2]	0x1008	0x1	0x00	hit	miss
load B[3]	0x200C	0x2	0x00	hit	hit

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	hit	hit
store A[1]	0x1004	0x1	0x00	hit	miss
load B[2]	0x2008	0x2	0x00	hit	hit
store A[2]	0x1008	0x1	0x00	hit	miss
load B[3]	0x200C	0x2	0x00	hit	hit
store A[3]	0x100C	0x1	0x00	hit	miss

Lines: 512
Sets: 256
Ways: 2
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- The performance of a 8K, 2-way, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	hit	hit
store A[1]	0x1004	0x1	0x00	hit	miss
load B[2]	0x2008	0x2	0x00	hit	hit
store A[2]	0x1008	0x1	0x00	hit	miss
load B[3]	0x200C	0x2	0x00	hit	hit
store A[3]	0x100C	0x1	0x00	hit	miss
	hit rate			75%	37.5%

Lines: 512
 Sets: 256
 Ways: 2
 Offset: 4
 index: 8
 tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
-----------	---------	-----	-------	-------------	----------

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	miss	hit

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	miss	hit
store A[1]	0x1004	0x1	0x00	miss	miss

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	miss	hit
store A[1]	0x1004	0x1	0x00	miss	miss
load B[2]	0x2008	0x2	0x00	miss	hit

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	miss	hit
store A[1]	0x1004	0x1	0x00	miss	miss
load B[2]	0x2008	0x2	0x00	miss	hit
store A[2]	0x1008	0x1	0x00	miss	miss

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	miss	hit
store A[1]	0x1004	0x1	0x00	miss	miss
load B[2]	0x2008	0x2	0x00	miss	hit
store A[2]	0x1008	0x1	0x00	miss	miss
load B[3]	0x200C	0x2	0x00	miss	hit

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	miss	hit
store A[1]	0x1004	0x1	0x00	miss	miss
load B[2]	0x2008	0x2	0x00	miss	hit
store A[2]	0x1008	0x1	0x00	miss	miss
load B[3]	0x200C	0x2	0x00	miss	hit
store A[3]	0x100C	0x1	0x00	miss	miss

Lines: 256
Sets: 256
Ways: 1
Offset: 4
index: 8
tag: 20

Write Allocate v.s. No - Write Allocate

- What if it's a 4K, direct-mapped, 16Byte blocked cache?
- How are the bits divided?

operation	address	tag	index	write-alloc	no-write
load B[0]	0x2000	0x2	0x00	miss	miss
store A[0]	0x1000	0x1	0x00	miss	miss
load B[1]	0x2004	0x2	0x00	miss	hit
store A[1]	0x1004	0x1	0x00	miss	miss
load B[2]	0x2008	0x2	0x00	miss	hit
store A[2]	0x1008	0x1	0x00	miss	miss
load B[3]	0x200C	0x2	0x00	miss	hit
store A[3]	0x100C	0x1	0x00	miss	miss
	hit rate			0%	37.5%

Lines: 256
 Sets: 256
 Ways: 1
 Offset: 4
 index: 8
 tag: 20

Write back v.s. Write Through

- Assume 8K, 2-way, 16Byte blocked cache with write-allocate policy
- Consider the following code

```
int A[1024]; /* A = 0x1000 */
int B[128]; /* B = 0x2000 */
int i;
for(i = 0 ; i < 1024; i++)
    B[i%128] += A[i];
```
- The all but the last 128 updates to B are temporary.
- WT: 1024 writes to L2
- WB: 128 writes to L2